

STORK and PENGUIN: Logic programming systems using general clauses and defaults

by R. A. de T. Guerreiro
A. S. Hemerly
M. A. Casanova

This paper describes two logic programming systems with the expressive power of full clausal first-order logic and with a nonmonotonic component. They provide a direct generalization of pure Prolog and can be implemented using the same technology as Prolog processors. The inference engine of both systems is based on the weak-model elimination method which, in the case of the second system, is extended to incorporate defaults.

1. Introduction

This paper first addresses the foundations of a class of logic programming systems that provide a direct generalization of pure Prolog to cope with full first-order logic and with nonmonotonic reasoning. Systems in this class can be implemented using the same technology as Prolog processors, as a consequence of the judicious choice of a variation of weak-model elimination as the basic refutation method and the adoption of defaults to capture nonmonotonic reasoning. Then, the paper outlines two specific systems that are completely operational.

More precisely, the first contribution of this paper consists in defining an adaptation of weak-model

elimination (WME) [1] that is sound and complete with respect to computing answers when the logic programs and queries are expressed by sets of generic clauses. The proofs of these two results are far more complex than the corresponding results for SLD resolution (see [2]), the basis of Prolog systems. The question of computing only definite answers is also settled, using a new result regarding refutations in WME. Such results provide the foundations for a class of logic programming systems with the expressive power of full first-order logic.

The second contribution of this paper is to adapt WME as the underlying refutation method of default logic [3] for the class of clausal defaults. Within this broader scope, the notion of an answer to a query posed to a logic program with defaults raises interesting questions that are briefly discussed. Defaults provide a much more flexible nonmonotonic formalism than negation by finite failure [4], the treatment of negation commonly adopted to extend the expressive power of logic programs and queries in Prolog.

Finally, this paper outlines two logic programming systems, STORK and PENGUIN, whose foundations follow from the results stated above. STORK is a full clausal first-order system supporting classical negation as well as negation by finite failure, whereas PENGUIN extends STORK with clausal defaults to capture

©Copyright 1992 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

nonmonotonic reasoning. The paper also discusses some extralogical facilities, which were carefully designed to extend the meaning they have in Prolog, or to deal with new programming issues arising in these logic programming systems.

STORK and PENGUIN are both coded in Prolog and are completely operational. The implementation started with the familiar idea of building Prolog interpreters in Prolog, adapted to the specific characteristics of model elimination. The Prolog code suffered successive optimizations until it achieved an adequate behavior in terms of space and memory. By using this strategy, the implementations could reuse all input-output and workspace management extralogical predicates of Prolog, among others.

A detailed account of the results reported here can be found in Casanova et al. [5] and Guerreiro et al. [6]. The logic programming systems are fully described in Silva et al. [7]. An early implementation of the original weak-model elimination method is described in Fleisig et al. [8], and a more recent implementation proposal, using the same principles as Prolog interpreters, is sketched in Stickel [9] and Casanova and Walter [10]. Other attempts to extend Prolog to the expressive level of full first-order logic can be found in Lloyd and Topor [11], Aida et al. [12], and Poole and Goebel [13]. A proposal for a nonclausal system with many inference rules is described in Bowen [14].

The organization of this paper is as follows. Section 2 first reviews the weak-model elimination method. It then recalls default logic, concentrating on a special class of defaults and using weak-model elimination as the underlying refutation method. Section 3 introduces the definitions of *program*, *query*, and *answer*. It centers on the question of computing answers, using generic clauses, and then incorporating defaults. Section 4 describes the two logic programming systems, and Section 5 contains the conclusions.

2. Preliminaries

◆ Weak-model elimination

Weak-model elimination (WME) has several attractive characteristics. First, it accepts as input sets of generic clauses, that is, sets of clauses with an arbitrary number of positive or negative literals. Moreover, WME is input-linear, does not use factoring, and yet is refutationally complete. To achieve completeness, the inference rules of WME sometimes maintain the resolved literals within the derived clauses and keep the literals (resolved or not) ordered within a clause. To distinguish these extended clauses from ordinary clauses, they will be called *chains*. Moreover, resolved literals in a chain will be enclosed within brackets.

More precisely, a *resolved literal* (or an *R-literal*) is an expression of the form $[L]$, where L is a literal. An *element* is a literal or an R-literal. An *elementary chain* is any sequence of literals, and a *chain* is any sequence of elements. The symbol $\square$ will denote the empty chain, which is elementary by definition. Each chain C represents, by convention, the universal closure of the disjunction of its literals, in the sense that any structure for the first-order alphabet in question satisfies C if and only if it satisfies the formula that C represents. Hence, the R-literals of a chain do not influence its semantics. From these definitions, it should be clear that elementary chains and clauses are one and the same concept and will be used interchangeably in this text.

The next definitions are basic for the inference rules of the method and assume familiarity with the notion of unification. In what follows, $B'B''$ denotes the concatenation of two chains B' and B'' , and $|L|$ indicates the atom of a literal L . Two literals L' and L'' can be *canceled by a substitution* θ if and only if θ is a most general unifier (mgu) of $\{|L'|, |L''|\}$ and $L'\theta$ and $L''\theta$ are complementary.

Let A' be a chain and let A'' be an elementary chain. Let β be a renaming of the variables of A'' such that $A''\beta$ has variables distinct from those of A' . Let L' be the leftmost element of A' , and suppose that L' is a literal. A chain A is an *extension* of A' by A'' (with mgu θ and renaming β) if and only if there exists a literal L'' of A'' and a substitution θ such that L' and $L''\beta$ can be canceled by θ and $A = B'B'$, where B'' is the chain $A''\beta\theta$ with the literal $L''\beta\theta$ removed and B' is a chain $A'\theta$ with the literal $L'\theta$ replaced by $[L'\theta]$.

Let A' be a chain. Let L' be the leftmost element of A' , and suppose that L' is a literal. A chain A is a *reduction* of A' (with mgu θ) if and only if there exists an R-literal $[M']$ of A' and a substitution θ such that L' and M' can be canceled by θ and A is $A'\theta$ with the literal $L'\theta$ removed.

A chain A is a *contraction* of a chain A' if and only if A is the chain obtained by removing from A' all R-literals that are to the left of the leftmost literal. If A has only R-literals, the result becomes the empty chain.

A chain A is a *full extension* of A' by A'' if and only if A is the contraction of an extension of A' by A'' . A chain A is a *full reduction* of a chain A' if and only if A is the contraction of a reduction of A' .

The weak-model elimination method works with the class of all sets of first-order chains; it has no axioms and two inference rules, *full extension* and *full reduction*, defined as follows:

Full extension: If A' and A'' are chains and A is a full extension of A' by A'' , then derive A from A' and A'' .

Full reduction: If A' is a chain and A is a full reduction of A' , then derive A from A' .

A WME deduction of a chain C from a set S of elementary chains is any finite sequence of chains $E = (E_1, \dots, E_n)$ such that C is the last chain of E , there is $i \leq n$ such that $E_1, \dots, E_i$, the *prefix* of E , consists of chains in S and, for each $j \in [i + 1, n]$, E_j is derived from E_{j-1} , the *parent chain* of E_j , by full reduction or full extension, in the latter case using an *auxiliary chain* from the prefix of E . The chain E_1 is called the *initial chain* of E . A WME refutation from a set of elementary chains S is a WME deduction of the empty chain from S .

The WME method defined above is slightly different from the original version of Loveland [1], but the results therein can be easily adapted to establish that WME is refutationally sound and complete.

• Clausal default logic

In this section, we first adapt the basic definitions of default logic [3] to a special class of normal defaults, called clausal defaults. Then we present a variation of Reiter's default proof theory for clausal defaults which is based on WME.

The *clausal default* is any expression of the form $A:C$, where the *prerequisite* A of the default is a conjunction of literals and the *consequent* C of the default is a clause. A *clausal default theory* is a pair $\Delta = (D, P)$, where P is a finite set of elementary chains and D is a finite set of clausal defaults. We will denote by $\text{consequent}(D)$ the set of the consequents of all defaults in D . We also accept $:C$ as a clausal default.

The clausal default $A:C$ should be understood as a convenient way of expressing the open normal default $A:MF/F$, in the notation of Reiter [3], where F is a disjunction of the literals in the clause C . Therefore, since we will limit ourselves to clausal defaults, we will be concerned with a particular case of open normal defaults.

The semantics for clausal default theories follows from the concept of extensions for open default theories. Then, given a clausal default theory $\Delta = (D, P)$, a clausal default $A:C$ in D should be interpreted as a generator of the set of defaults $A\theta:C\theta$, for all substitutions θ of the variables in A and C by terms of the Herbrand universe for the current alphabet. The clausal default $A\theta:C\theta$ therefore reads: $C\theta$ can be assumed "by default" if the prerequisite $A\theta$ is believed and $C\theta$ is consistent with the beliefs.

Let $\Delta = (D, P)$ be a clausal default theory and let Q be a disjunction of conjunctions of literals. Let $CL(\neg\exists Q)$ denote a clausal representation of the negation of the existential closure of Q . Intuitively, a refutation with defaults from Δ and Q is a sequence of WME refutations such that the first is a WME refutation from the chains in $P \cup CL(\neg\exists Q) \cup \text{consequent}(D)$ with initial chain in $CL(\neg\exists Q)$ and, after the first, each WME refutation intends to refute the negation of the conjunction of the prerequisites of the defaults used on the preceding WME

refutation, with the appropriate substitutions, from the chains in $P \cup \text{consequent}(D)$ [but not in $CL(\neg\exists Q)$]. The sequence should also satisfy a global consistency test, verifying whether the use of the defaults is acceptable.

To record the substitutions affecting each use of each default in each WME refutation in the sequence, we will use default literals. This section then redefines the notion of chain and the inference rules of WME to register such substitutions. An *indexed chain* is a pair of the form (C, N) , where C is a chain and N is a set of literals. The *indexing* of $\Delta = (D, P)$ is the set of pairs of the form $(C_i, \emptyset)$, where $C_i \in P$, or the form $(C_i, \{\delta_i(\bar{x}_i)\})$, for each default $A_i:C_i$ in D , where $\bar{x}_i$ is a list of the variables occurring in A_i and C_i and δ_i is a new predicative symbol whose arity is the length of $\bar{x}_i$, called a *default literal*. The *indexing* of Q consists of the set of pairs $(C_i, \emptyset)$, where $C_i \in CL(\neg\exists Q)$. The default literal $\delta_i(\bar{x}_i)$ will record the substitutions applied to the variables of the default $A_i:C_i$. But to effect this recording, the inference rules of WME had to be modified, as we now describe.

An indexed chain (A, L) is a *full indexed reduction* of an indexed chain (A', L') if and only if A is a full reduction of A' with mgu θ and $L = L'\theta$. An indexed chain (A, L) is a *full indexed extension* of (A', L') by an elementary indexed chain (A'', L'') if and only if A is a full extension of A' by A'' , with mgu θ and renaming β of A'' , and $L = L'\theta \cup L''\beta\theta$.

Let R be an indexed WME refutation from the chains in the indexing of Δ and Q . Suppose that R terminates in $(\square, S)$. A default ψ is *returned* by R iff there is a pair $(C_i, \{\delta_i(\bar{x}_i)\})$ in the indexing of Δ , corresponding to some default $A_i:C_i$ in D , and there exists a literal of the form $\delta_i(\bar{t})$ in S such that ψ is $A_i\theta:C_i\theta$ where $\theta = \{\bar{x}_i/\bar{t}\}$.

Let $A_i:C_i$ be a default in D and $(C_i, \{\delta_i(\bar{x}_i)\})$ be the corresponding pair in the indexing of Δ . This default is *fired* in R iff there exists an indexed chain in R derived by indexed full extension with $(C_i, \{\delta_i(\bar{x}_i)\})$ as auxiliary chain. Then, each default $A_i:C_i$ fired in R , as well as each default corresponding to a default literal in the initial chain of R , if any, generates a *descendent* default in the set of defaults returned by R .

A *candidate WME refutation sequence with defaults* from $\Delta = (D, P)$ and Q is a finite sequence $R = (R_0, \dots, R_k)$ of indexed WME refutations such that

1. R_0 is an indexed WME refutation from the indexing of Δ and the indexing of Q , with initial chain in indexing of Q .
2. For $0 \leq i \leq k$, let D_i be the set of defaults returned by R_i ; M_i be the set of default literals corresponding to the defaults in D_i ; D^0 be the set of defaults in D_i which are the descendents of the defaults fired in R_i ; and B_i be the chain representing the negation of the conjunction of the prerequisites of all defaults in D^0 . Then,
 - (a) For $1 \leq i \leq k$, R_i must be an indexed WME refutation, with initial chain $\{(B_{i-1}, M_{i-1})\}$ from the

1. program(a,fortran)
2. program(b,pascal)
3. program(c,fortran) program(c,pascal)
4. calls(a,b)
5. calls(b,c)
6. $\neg \text{calls}(x,y) \text{ depends}(x,y)$
7. $\neg \text{calls}(x,z) \neg \text{depends}(z,y) \text{ depends}(x,y)$

(1)

indexed chains in the indexing of Δ together with the pair (B_{i-1}, M_{i-1}) .

(b) $D^{(k)} = \emptyset$.

A WME refutation sequence with defaults from $\Delta = (D, P)$ and Q is a candidate WME refutation sequence with defaults that satisfies the following additional condition:

Consistency test Let C be the set of consequents of all defaults occurring in D_k . If P is satisfiable, there is a substitution θ of the variables occurring in C by ground terms of the Herbrand universe over the current alphabet such that $P \cup C\theta$ is satisfiable.

It follows from the results in Reiter [3] and from the soundness and completeness theorem for computing definite answers (see "Computing answers with WME" below) that the WME method, adapted to account for clausal defaults as described above, is refutationally correct and complete.

When using the results in Reiter [3], one must, however, take into account the following observations. Recall that the class of *top-down default proofs* in [3] is defined as a restriction of the class of *admissible refutation sequences*, defined very similarly to our sequences. The restriction is actually a test that relates the descendants of the defaults fired in each refutation of an admissible refutation sequence. In our definition of the WME refutation sequence with defaults, this test can be ignored for the following reasons.

First, recall that the test becomes necessary in two cases: 1) when a derived clause is reused as an auxiliary clause in one of the refutations in the sequence; 2) when a refutation in the sequence uses more than once clauses originating from the negation of the prerequisites of the defaults fired in the previous refutation. However, in our approach, the first case never holds because WME is input-linear, and the second case can be avoided by the use of clausal defaults. Indeed, the negation of the conjunction of the prerequisites of the defaults fired in a refutation of a WME refutation sequence with defaults maps into a single

chain that can be viewed as a query. Hence, the second case of the test reduces to computing definite answers to this query, which we know how to do through the variation of WME that is presented at the end of the subsection on computing answers with WME and has been directly incorporated into our definition of WME refutation sequences with defaults [condition 2(a)].

3. Computing answers

• Programs, queries, and answers

Recall that a *program* P is a finite set of clauses and a *query* Q is a disjunction of conjunctions of literals, that is, a quantifier-free formula in disjunctive normal form. A query is *definite* iff it is a single conjunction of literals; otherwise it is *indefinite*.

An *answer* A to a query Q over a program P is either **False** or a disjunction of instances of conjunctions in Q over the alphabet of P and Q , that is, a disjunction of conjunctions obtained from those in Q by substituting variables by terms of the alphabet used to write P and Q . An answer is *definite* iff it consists of a single conjunction; otherwise it is *indefinite* [15].

An answer A to Q over P is *correct* iff P logically implies $\forall A$, the universal closure of A . Finally, an answer A to Q over P is *more general* than an answer B to Q over P iff $\forall A$ logically implies $\forall B$. We let **False** be an answer simply because it will be the most general answer to any query over an inconsistent program.

For example, the set of clauses shown in Box 1 is a program, which we call **DIC**, in which clause 3 indicates that c is an ordinary program written in FORTRAN or Pascal and clauses 6 and 7 indicate that x depends on y if x calls y directly or transitively. The formula below is a query, which we call **DEP[a]**:

$$(\text{depends}(a,x) \wedge \text{program}(x,\text{pascal})) \vee (\text{depends}(a,x) \wedge \text{program}(x,\text{fortran})).$$

It asks for a program written in FORTRAN or Pascal on which program a depends. An answer A to **DEP[a]** over **DIC** would be

$$\text{depends}(a,b) \wedge \text{program}(b,\text{pascal}).$$

Indeed, the conjunction in A is an instance of the first conjunction in **DEP[a]**. It is in fact a correct answer because **DIC** logically implies $\forall A$. A second correct answer to **DEP[a]** over **DIC** would be

$$(\text{depends}(a,c) \wedge \text{program}(c,\text{fortran})) \vee (\text{depends}(a,c) \wedge \text{program}(c,\text{pascal})).$$

Therefore, an indefinite query may have both indefinite and definite answers.

As another example, the formula which follows is also a query (call it **Lang**):

program(c,x).

It asks for the language in which program c is written. It has only one correct answer, which is

program(c,fortran) $\vee$ program(c,pascal).

Therefore, a definite query may have indefinite answers. It is also possible that a definite query may have both indefinite and definite answers.

A *program with defaults* is a pair $\Delta = (\mathbf{D}, \mathbf{P})$, where $\mathbf{P}$ is a finite set of elementary chains and $\mathbf{D}$ is a finite set of clausal defaults. The definitions of query and answer are not modified. The definition of correct answer is now based on the concept of extensions of default theories, as we shall see. We present here a simple example for illustration.

For example, the set of clauses and defaults shown in Box 2 is a program in which clause 4 represents the default $\text{bird}(y):\text{fly}(y)/\text{fly}(y)$, which means that "if y is a bird and it is consistent to assume that y flies, then y flies." The formula $\text{fly}(x)$ is a query for the program above. The correct answer to this query is $\text{fly}(\text{canary})$.

• Computing answers with WME

Recall that $CL(\neg\exists\mathbf{Q})$ denotes the clausal representation of the negation of the existential closure of a query $\mathbf{Q}$. Given a WME refutation R from the elementary chains in a program $\mathbf{P}$ and in $CL(\neg\exists\mathbf{Q})$, it is possible to show that the substitutions applied to the free variables of chains in $CL(\neg\exists\mathbf{Q})$ during the construction of R induce a correct answer to $\mathbf{Q}$ over $\mathbf{P}$. However, to recover such substitutions is not exactly simple, because $CL(\neg\exists\mathbf{Q})$ may possibly contain more than one chain, and each of those chains may be used more than once in R . This section then introduces answer literals to register such substitutions [16].

An *activated chain* is a pair of the form $(\mathbf{C}, \mathbf{L})$, where $\mathbf{C}$ is a chain and $\mathbf{L}$ is a set of literals. The *activation* of $\mathbf{P}$ is the set $\text{activate}(\mathbf{P})$ consisting of the activated chains $(\mathbf{C}, \emptyset)$, where $\mathbf{C} \in \mathbf{P}$. The *activation* of a query $\mathbf{Q}$ of the form $\mathbf{Q}_1 \vee \dots \vee \mathbf{Q}_n$ is the set $\text{activate}(\mathbf{Q})$ of activated chains $(\sim\mathbf{Q}_i, \{r_i(\bar{x}_i)\})$, $i = 1, \dots, n$, where $\sim\mathbf{Q}_i$, by convention, is the chain consisting of the complement of the literals of $\mathbf{Q}_i$, $\bar{x}_i$ is a list of the variables of $\mathbf{Q}_i$, and r_i is a predicate symbol, not in the original alphabet, whose arity is equal to the length of $\bar{x}_i$. The literal $r_i(\bar{x}_i)$ is the *answer literal* for $\mathbf{Q}_i$ in the activation of $\mathbf{Q}$.

The activation of a query $\mathbf{Q}$ therefore produces a clausal representation of the negation of the existential closure of $\mathbf{Q}$, with each elementary chain annotated with an answer literal whose function is to record the substitutions applied to the variables of the chain. For that purpose, the inference rules of WME and the notions of WME deduction and WME refutation are also modified to

- | | |
|------------------|-------------------------------------|
| 1. bird(penguin) | 3. $\neg\text{fly}(\text{penguin})$ |
| 2. bird(canary) | 4. $\text{bird}(y):\text{fly}(y)$ |

(2)

account for answer literals in a manner similar to that described in Section 2 for default literals.

An answer $\mathbf{A}$ to $\mathbf{Q}$ over $\mathbf{P}$ is *WME-computed* if and only if there is an activated WME refutation R from $\text{activate}(\mathbf{P}) \cup \text{activate}(\mathbf{Q})$ such that either R terminates in $(\square, \emptyset)$, in which case $\mathbf{A}$ must be equal to **False**, or R terminates in $(\square, \mathbf{L})$, with $\mathbf{L} \neq \emptyset$, and $\mathbf{A}$ is a disjunction of all conjuncts $\mathbf{B}$ such that there is $(\sim\mathbf{Q}_i, \{r_i(\bar{x}_i)\}) \in \text{activate}(\mathbf{Q})$ and $r_i(\bar{t}) \in \mathbf{L}$ such that $\mathbf{B}$ is equal to $\mathbf{Q}_i\theta$, where $\theta = \{\bar{x}_i/\bar{t}\}$.

The following example illustrates how the method computes a definite answer to an indefinite query. Consider again the program **DIC** and the query **DEP[a]** introduced in Section 2. An activated WME refutation from the set of chains in the activation of **DIC** and **DEP[a]** is shown in Box 3. Hence, $\mathbf{A} = \text{depends}(a,b) \wedge \text{program}(b,\text{pascal})$ is a WME-computed answer to **DEP[a]** over **DIC**, since $r_2(b)$ in step 12 indicates that b was substituted for the variable v of the chain in step 9.

The WME method, modified as described above, is sound and complete for computing answers in the following sense:

Theorem 1 (Soundness and Completeness Theorem) [17]

Let $\mathbf{P}$ be a program and $\mathbf{Q}$ be a query.

- Every WME-computed answer to $\mathbf{Q}$ over $\mathbf{P}$ is correct.
- Given any correct answer $\mathbf{A}$ to $\mathbf{Q}$ over $\mathbf{P}$, there is a WME-computed answer which is more general than $\mathbf{A}$.

We will conclude this section with another variation of weak-model elimination that computes only definite answers.

Let $\mathbf{S}$ be a set of activated elementary chains and $\mathbf{T}$ be a subset of $\mathbf{S}$. We say that an activated WME refutation R from $\mathbf{S}$ has *initial support* from $\mathbf{T}$ iff the initial activated chain of R is in $\mathbf{T}$ and no activated chain in $\mathbf{T}$ is ever used as an auxiliary chain in derivations in R .

Let $\mathbf{Q}$ be a query to a program $\mathbf{P}$. An answer $\mathbf{A}$ to $\mathbf{Q}$ over $\mathbf{P}$ is *WME-computed with initial support from $\mathbf{Q}$* iff there is an activated WME refutation R from $\text{activate}(\mathbf{P}) \cup \text{activate}(\mathbf{Q})$, with initial support from $\text{activate}(\mathbf{Q})$, that computes $\mathbf{A}$. Note that, since just one chain from $\text{activate}(\mathbf{Q})$ is used in R , $\mathbf{A}$ is a definite answer. In fact, we can prove the following.

- | | |
|---|------------------------|
| 1. (program(a,fortran), $\emptyset$) | . activation of DIC |
| 2. (program(b,pascal), $\emptyset$) | . activation of DIC |
| 3. (program(c,fortran) program(c,pascal), $\emptyset$) | . activation of DIC |
| 4. (calls(a,b), $\emptyset$) | . activation of DIC |
| 5. (calls(b,c), $\emptyset$) | . activation of DIC |
| 6. ($\neg$ calls(x,y) depends(x,y), $\emptyset$) | . activation of DIC |
| 7. ($\neg$ calls(x,z) $\neg$ depends(z,y), depends(x,y), $\emptyset$) | . activation of DIC |
| 8. ($\neg$ depends(a,u) $\neg$ program(u,fortran), $\{r_1(u)\}$) | . activation of DEP[a] |
| 9. ($\neg$ depends(a,v) $\neg$ program(v,pascal), $\{r_2(v)\}$) | . activation of DEP[a] |
| 10. ($\neg$ calls(a,y) [$\neg$ depends(a,y)] $\neg$ program(y,pascal), $\{r_2(y)\}$) | . full ext. of 9 by 6 |
| 11. ($\neg$ program(b,pascal), $\{r_2(b)\}$) | . full ext. of 10 by 4 |
| 12. ($\square$, $\{r_2(b)\}$) | . full ext. of 11 by 2 |

(3)

Theorem 2 (Soundness and Completeness Theorem for Definite Answers) [17]

Let **P** be a program and **Q** be a query.

- (a) Let **A** be an answer to **Q** over **P** that is WME-computed with initial support from **Q**. Then, **A** is definite and correct.
- (b) Given any definite correct answer **A** to **Q** over **P**, there is a definite answer **B** to **Q** over **P** such that **B** is WME-computed with initial support from **Q** and **B** is more general than **A**.

• *Computing answers with clausal defaults*

The computation of answers now combines two notions described in preceding subsections: activation and indexing. We show that the inference rules now work with triples of the form (C, L, M) , where **C** is a chain, **L** is a set of answer literals for computing answers of a query, following the description above, and **M** is a set of default literals for monitoring the defaults used in a refutation, as described in Section 2. Note that the answer literals record all of the substitutions applied to variables of chains from the query during the construction of each refutation in a WME refutation sequence with defaults and in the consistency test.

The definition of the WME refutation sequence with defaults immediately induces the following notion of computed answer. Let $\Delta = (D, P)$ be a program with defaults and **Q** be a query to Δ . An answer **A** to **Q** over Δ is *WME-computed by defaults* iff there exists a WME refutation sequence with defaults **R** from the activation and indexing of Δ and **Q** such that the last refutation in **R** terminates with $(\square, S, E)$; the consistency test for **R** uses the substitution θ ; and either **S** = $\emptyset$, in which case **E** = $\emptyset$ and **A** must be equal to **False**, or **S** $\neq \emptyset$, and **A** is a

disjunction of all conjuncts **B** such that there is $(\neg Q_i, \{r_i(\bar{x}_i)\}, \emptyset)$ in the activation and indexing of **Q** and there is $r_i(\bar{t}) \in S$ such that **B** is equal to $Q_i \gamma \theta$, where $\gamma = \{\bar{x}_i/\bar{t}\}$.

This definition is not reasonable, because it admits answers with arbitrary instantiations, coming from the substitution θ generated for the consistency test. On the other hand, by the semantics itself of open defaults, it is not possible to abandon such substitution under the risk of invalidating the correctness of WME refutation sequences with defaults. For example, let $\Delta = (D, P)$ be a program, where $D = \{\text{bird}(y):\text{fly}(y)\}$ and $P = \{\text{bird}(z), \neg\text{fly}(\text{penguin}), \neg\text{fly}(\text{ostrich}), \text{yellow}(\text{canary})\}$. Consider the query **Q** = $\text{fly}(x)$. Consider the WME refutation sequence with defaults $R = (R_0, R_1)$, from Δ and **Q**, constructed as follows. First, R_0 is an indexed and activated WME refutation from the activation and indexing of Δ and **Q** shown in Box 4.

Note that R_0 returns the default $\text{bird}(x):\text{fly}(x)$. Hence, R_1 must be an indexed and activated WME refutation from the chains in the indexing and activation of Δ and the chain representing the negation of the prerequisite of $\text{bird}(x):\text{fly}(x)$ (chain 1.2 in Box 5). Note that chain 1.2 carries on the set of answer literals and the set of default literals from chain 0.3. This is necessary to correctly compute answers.

By the definition of the WME refutation sequence with defaults, we must also test the consistency of the set $E = \{\text{bird}(z), \neg\text{fly}(\text{penguin}), \neg\text{fly}(\text{ostrich}), \text{yellow}(\text{canary})\} \cup \{\text{fly}(x)\theta\}$ for some substitution θ of x by a term of the Herbrand universe of the alphabet in question. Indeed, by taking $\theta = \{x/\text{canary}\}$, the set **E** becomes consistent. Hence, $\text{fly}(\text{canary})$ is the answer WME-computed by the refutation **R**, for this choice of θ .

Note that the choice of θ is entirely arbitrary. On the other hand, it is not possible to ignore θ , since the set

$\{\text{bird}(z), \neg\text{fly}(\text{penguin}), \neg\text{fly}(\text{ostrich}), \text{yellow}(\text{canary})\} \cup \{\text{fly}(x)\}$ is not satisfiable. Intuitively, the default in **D** can not be fired for a substitution of x by, for example, penguin.

To solve the above dilemma, we propose to always base the consistency test on a class of substitutions that change each variable by a new constant not appearing in **P**, **D**, and **Q**, whose intuitive semantics would be “the typical individual such that. . .” In the current example, we introduce the new constant p_0 , understood as “the typical bird.” Consider again the refutation with defaults R , except that the substitution of the consistency test is now, by definition, $\theta = \{x/p_0\}$. Since, for this choice of θ , the set $\{\text{bird}(z), \neg\text{fly}(\text{penguin}), \neg\text{fly}(\text{ostrich}), \text{yellow}(\text{canary})\} \cup \{\text{fly}(x)\theta\}$ is consistent, we have that $\text{fly}(p_0)$ is the new computed answer by the refutation R . Intuitively, this answer indicates that “the typical bird” flies. Note that the introduction of p_0 is similar to the Skolemization of the formula $\exists x(\text{fly}(x))$, except for the intuitive interpretation of the Skolem constant introduced (i.e., the typical element).

In order to formalize these concepts, we first redefine the answer to extend it to programs with defaults. An *answer A* to a query **Q** over a program $\Delta = (\mathbf{D}, \mathbf{P})$ is either **False** or a disjunction of instances of conjunctions in **Q** over the alphabet of Δ and **Q**, that is, a disjunction of conjunctions obtained from those in **Q** by substituting terms for variables over the alphabet in question.

We now formally define relativized answers and then relativized answers computed by defaults. Let $\Delta = (\mathbf{D}, \mathbf{P})$ be a program with defaults and **Q** be a query to Δ . Let $\mathcal{L}$ be the first-order language used.

We will work with relations in the Herbrand universe. An *answer A*, *relativized* by a relation **R** on the Herbrand universe of the alphabet in question, to the query **Q** over the program Δ is either **False** or an expression of the form $\Lambda \bar{y}(\mathbf{R} \Rightarrow \mathbf{K})$, where **K** is an answer to the query **Q** over the program Δ and $\bar{y}$ is a list of some of the variables of **Q** with the same length as the arity of **R**. The variables in $\bar{y}$ are called the *connection variables*. The intended meaning of such an expression is “If $\bar{e} \in \mathbf{R}$, then $\mathbf{K}\{\bar{y}/\bar{e}\}$.”

An answer **A** to **Q** over Δ is *correct* iff there is an extension E of Δ such that E logically implies $\forall \mathbf{A}$. This definition extends to relativized answers. A relativized answer $\Lambda \bar{y}(\mathbf{R} \Rightarrow \mathbf{K})$ to **Q** over Δ is *correct* iff, for every element $\bar{e}$ of **R**, there is an extension E of Δ such that E logically implies $\forall (\mathbf{K}\{\bar{y}/\bar{e}\})$.

We say that a Herbrand structure **H** is a *model* for an open formula **F** iff **H** is a model for $\forall \mathbf{F}$. A Herbrand structure **H** is a *model* for a relativized answer $\Lambda \bar{y}(\mathbf{R} \Rightarrow \mathbf{K})$ iff **H** is a model for $\mathbf{K}\{\bar{y}/\bar{e}\}$, for each $\bar{e} \in \mathbf{R}$.

Given two answers (respectively relativized answers) **A**₁ and **A**₂ to **Q** over Δ , we say that **A**₁ is *more general than* **A**₂ iff every Herbrand model of **A**₁ is also a model of **A**₂.

Assume now that $\mathcal{L}$ contains an enumerable family of constants. A *relativized answer A* to **Q** over Δ is *computed*

- | | |
|---|------------------------------|
| 0.1 $(\text{fly}(y), \emptyset, \{\delta(y)\})$ | (originating from Δ) |
| 0.2 $(\neg\text{fly}(x), \{r(x)\}, \emptyset)$ | (originating from Q) |
| 0.3 $(\square, \{r(x)\}, \{\delta(x)\})$ | (extension of 0.2 by 0.1) |

(4)

- | | |
|---|---|
| 1.1 $(\text{bird}(z), \emptyset, \emptyset)$ | (from Δ) |
| 1.2 $(\neg\text{bird}(x), \{r(x)\}, \{\delta(x)\})$ | (from the negation of the prerequisite) |
| 1.3 $(\square, \{r(x)\}, \{\delta(x)\})$ | (extension of 1.2 by 1.1) |

(5)

by defaults iff there exists a candidate WME refutation sequence with defaults $(R_1, \dots, R_k)$ from the activation and indexing of Δ and **Q** such that the last refutation in R_k ends in $(\square, \mathbf{S}, \mathbf{E})$, **C** is the set of consequents of all defaults returned by R_k , and either $\mathbf{S} = \emptyset$, in which case **A** must be equal to **False**, or $\mathbf{S} \neq \emptyset$ and **A** is of the form $\Lambda \bar{y}(\text{CONSIST}[\mathbf{P}, \mathbf{C}] \Rightarrow \mathbf{K})$, where **K** is a disjunction of all conjunctions **B** such that there is $(\sim \mathbf{Q}_i, [r_i(\bar{x})], \lambda)$ in the activation and indexing of **Q** and there is $r_i(\bar{t}) \in \mathbf{S}$ such that **B** is equal to $\mathbf{Q}_i\gamma$, where $\gamma = \{\bar{x}/\bar{t}\}$, and $\bar{y}$ is the list of all variables in **C**. We define $\text{CONSIST}[\mathbf{P}, \mathbf{C}]$ as the relation in the Herbrand universe of $\mathcal{L}$ as follows:

$\bar{e} \in \text{CONSIST}[\mathbf{P}, \mathbf{C}]$ iff $\mathbf{P} \cup \mathbf{C}\{\bar{y}/\bar{e}\}$ is consistent.

The intuitive semantics of $\Lambda \bar{y}(\text{CONSIST}[\mathbf{P}, \mathbf{C}] \Rightarrow \mathbf{K})$ is that “for all tuples of terms $\bar{e}$ in the Herbrand universe, if $\bar{e} \in \text{CONSIST}[\mathbf{P}, \mathbf{C}]$, then $\mathbf{K}\{\bar{y}/\bar{e}\}$ is an answer.”

Given a relativized answer $\Lambda \bar{y}(\text{CONSIST}[\mathbf{P}, \mathbf{C}] \Rightarrow \mathbf{K})$, computed by defaults, the tuples in $\text{CONSIST}[\mathbf{P}, \mathbf{C}]$ are called *typical elements*. Assume that $\text{CONSIST}[\mathbf{P}, \mathbf{C}]$ has arity n ; then, the n -tuples over the Herbrand universe that are not typical elements are called *atypical elements*.

Consider again our previous example, where $\mathbf{P} = \{\text{bird}(z), \neg\text{fly}(\text{penguin}), \neg\text{fly}(\text{ostrich}), \text{yellow}(\text{canary})\}$, $\mathbf{D} = \{\text{bird}(y):\text{fly}(y)\}$, and **Q** = fly(x). The relativized answer computed by defaults is $\Lambda x(\text{CONSIST}[\mathbf{P}, \{\text{fly}(x)\}] \Rightarrow \text{fly}(x))$. Note that this relativized answer is more general than, for example, fly(canary). Actually, for this relativized answer, canary is a typical element and penguin is an atypical element.

Theorem 3 (Soundness and Completeness Theorem for Relativized Answers computed by defaults) [18]

Let $\mathcal{L}$ be the first-order logic language, $\Delta = (\mathbf{D}, \mathbf{P})$ be a program with defaults, and **Q** be a query to Δ . Assume that $\mathcal{L}$ contains an enumerable family of constants.

```

/*      Program ROMAN      */
1. man(marcus).
2. pompeian(marcus).
3. ruler(caesar).
4. tryassassinate(marcus,caesar).
5. hate(marcus,otavio).
6.  $\neg$ pompeian(X) | roman(X).
7.  $\neg$ roman(Y) | loyalto(Y,caesar) | hate(Y, caesar).
8. loyalto(Z, f(Z)).
9.  $\neg$ man(X) |  $\neg$ tryassassinate(X,Y) |  $\neg$ emperor(Y) |  $\neg$ loyalto(X,Y)

(6)

```

- (a) Every relativized answer computed by defaults to **Q** over Δ is correct.
- (b) Given any correct answer to **Q** over Δ , there is a relativized answer computed by defaults which is more general.

Finally, we observe that the consistency test sanctioning a refutation sequence with defaults can be implemented as a test for the finite failure to refute the consequents, together with the original set of clauses from the program.

4. A short description of two logic programming systems

• The STORK system

STORK is a full first-order logic system supporting classical negation as well as negation by finite failure, and offering a collection of extralogical predicates, patterned after those defined for Prolog, or designed to deal with programming aspects arising only in general-clause logic programming. STORK extends both the declarative and the operational semantics of Prolog, including the extralogical features, in such a way that a Prolog program retains its original meaning in STORK.

Syntax of the basic STORK language

The definitions of a STORK *atom*, *constant*, *variable*, *term*, and *atomic formula* or *predicate* are as for the basic Prolog language. A STORK *clause* is a list of atomic formulas, negated or not, separated by the symbol |. A STORK *program* is a sequence of STORK clauses, each one ending with a period (.). We also allow the inclusion of comments in a STORK program between the delimiters /* and */.

An *admissible query* is either a predicate or an expression of the form (Q), $\neg$ Q, P | Q, or P & Q, where P and Q are admissible queries. A STORK *query* is an admissible query followed by a period (.).

An *answer* is a disjunction of conjunctions originated from the query by substitutions. The priority among the

connectives maintains the following order, from the higher to the lower weight: $\neg$, &, |. The programmer can use “(” and “)” to modify this order.

Box 6 shows an example of a STORK program. For instance, the intended meaning of clause 7 is that “If Y is Roman, then either Y is loyal to Caesar or Y hates Caesar,” and that of clause 9 is “If X is a man, X tries to assassinate Y, and Y is an emperor, then X is not loyal to Y.”

A STORK query over the program ROMAN is

hate(marcus,X).

This query has two correct answers:

hate(marcus,otavio).
hate(marcus,caesar).

Operational semantics of the basic STORK language

This section introduces an operational semantics for the STORK language by defining the STORK abstract machine.

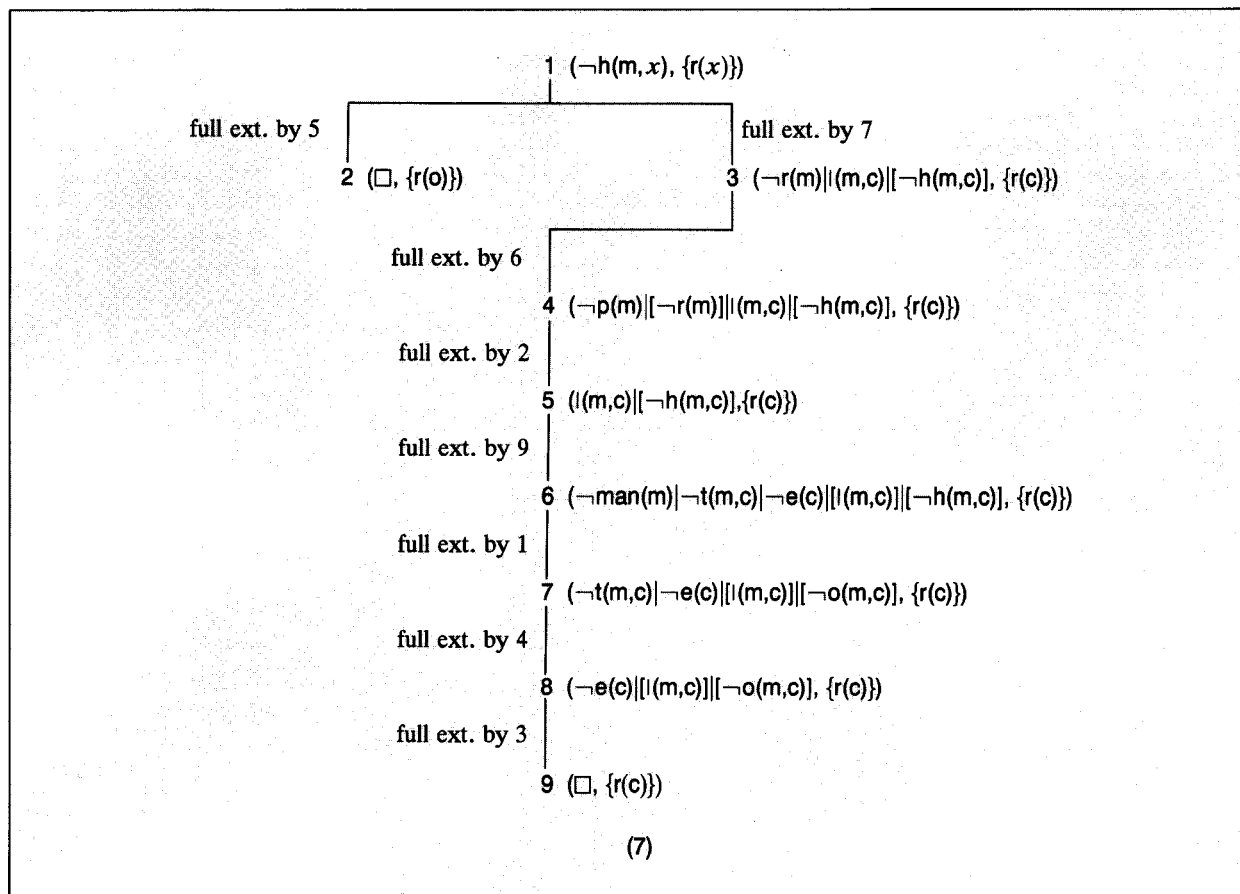
Given a STORK program **P** and a STORK query **Q**, the machine initially maps the STORK program clauses into a list of elementary chains. Then, the machine adds to the end of this list the set of the chains originating from $CL(\neg\exists\mathbf{Q})$. The machine constructs, in pre-order, the set of all refutation trees (based on WME) from these chains, whose roots are the chains originating from $CL(\neg\exists\mathbf{Q})$.

The machine will stop when it reaches a successful branch, returning the correct answer corresponding to this branch, or when all trees are built, when it returns **Fail**. However, the machine may produce no answer if it traverses an infinite branch.

The construction of each refutation tree observes the following rules:

- The full-reduction rule is exhaustively applied before the full-extension rule.
- In the application of the full-reduction rule, the R-literals in the chain in question are selected from left to right.
- In the application of the full-extension rule,
 - The entry chains are selected in the order in which they appear in the program.
 - The literals in each entry chain are selected from left to right.

For example, let **Q** be the query hate(marcus,X) over the program ROMAN. Note that the clausal representation of the negated query generates only one activated chain, ($\neg$ hate(marcus,X), {r(x)}), where r(x) is the associated answer literal. Then, the STORK machine will construct just one WME refutation tree [see Box 7, in which atoms are denoted by their initial letters, except for the predicate man to distinguish it from marcus, and nodes are numbered in the order they are generated]. Note that the



tree has two success branches. Therefore, the machine will return the answer corresponding to the first success branch detected, i.e., `hate(marcus,otavio)`. If requested, the machine will return the other correct answer, `hate(marcus,caesar)`.

Some extralogical facilities of STORK

STORK borrows several extralogical predicates directly from Prolog, such as arithmetic predicates, comparison predicates, input/output predicates, workspace management predicates, and debugging predicates. In particular, STORK has the metapredicate `¬%`, for negation by finite failure, the extralogical predicate `cut`, and the metapredicate `call`, which in STORK also correctly handles indefinite answers. But STORK also uses extralogical predicates specially designed to deal with programming aspects arising only in general-clause logic programming. As an example of a new predicate, we have the metapredicate `@`, which is similar to `call`, but computes only definite answers. For reasons of brevity, we discuss here only the `cut` and the `call` predicates, establishing a parallel between them and the corresponding Prolog predicates.

The cut predicate The extralogical predicate `cut`, denoted `/`, is the main control predicate of the backtracking operation. `Cut` behaves as a predicate which simply succeeds on being called. However, if backtracking later returns to the `cut`, the system discontinues the search in the subtree whose root is the node immediately superior. Thus, the `cut` causes the remainder of that subtree to be pruned from the WME refutation tree.

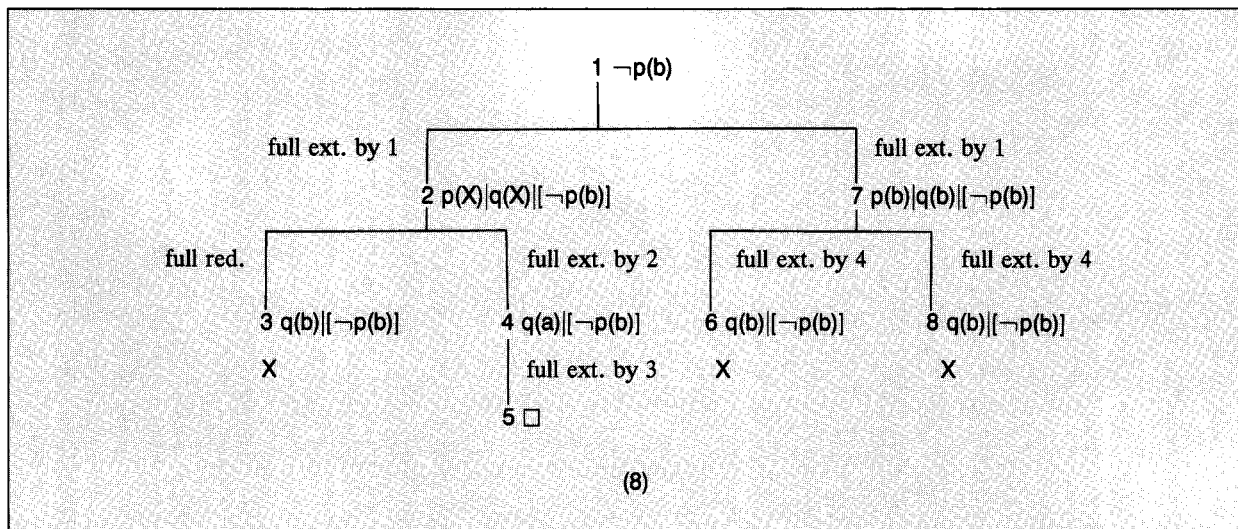
Note that the effects, in a Prolog program and in the corresponding STORK program, of the Prolog `cut` and the STORK `cut` are similar. For example, consider the following Prolog program and its corresponding STORK program:

<code>a ← / & b.</code>	<code>a / ¬b.</code>
<code>a.</code>	<code>a.</code>

Then, considering the same goal (query) for both:

<code>←a.</code>	<code>a.</code>
<code>GOAL : ← a.</code>	<code>Fail</code>
<code>OMS FAIL</code>	

The `cut` in the first clause blocks backtracking, the second clause is not evaluated, and both machines return **Fail**.



The processing of the STORK and Prolog machines is in fact identical for Horn clause programs which do and do not contain *cut*. However, in general, the STORK *cut* possibly prunes more branches than the Prolog *cut*, simply because the STORK machine has the option of applying two distinct inference rules, full extension or full reduction, and may select more than one literal to apply them.

The next example illustrates how the STORK *cut* may prune successful branches just because the strategy of the STORK machine is to apply all possible reductions before any extensions. Thus, as in Prolog, *cut* must be used with care.

Consider the following program and query:

1. $p(b) \mid p(X) \mid \mid q(X).$
2. $\neg p(a).$
3. $\neg q(a).$

and query: $p(b)$. Its negation is

4. $\neg p(b).$

A refutation tree, not considering the *cut*, is shown in Box 8. When the *cut* is taken into account, the successful branch ending on node 5, and those to the right of it, are not evaluated. Therefore, the STORK machine returns **Fail** and skips the computation of a correct answer.

The *call(C)* predicate The *call(C)* predicate evaluates *C* as a subquery. The argument *C* must be equivalent to a conjunction of predicates when *call(C)* is actually executed. More precisely, during the construction of a tree, if the chain labeling node *n* is of the form $\text{call}(C) \mid L_1 \mid \dots \mid L_m$, the STORK machine will call itself recursively, having as arguments the original program and

the query *C*. However, recall that STORK, unlike Prolog, may compute an answer which is a disjunction, even if the query is a simple conjunction. Hence, each answer to the recursive call will have the generic form $C\theta_1 \mid \dots \mid C\theta_n$, where θ_k , $1 \leq k \leq n$, is the substitution applied to the conjunction equivalent to *C*. Therefore, for each such answer, the machine will generate a new descendent of *n* labeled with $L_1\theta_1 \mid \dots \mid L_m\theta_1 \mid \dots \mid L_1\theta_n \mid \dots \mid L_m\theta_n$.

For example, consider the following program:

$q(a).$
 $q(b).$
 $p(a) \mid p(b).$

Then,

$\text{addcl}(q(a)).$
 Answer: $\text{addcl}(q(a)).$
 $\text{addcl}(q(b)).$
 Answer: $\text{addcl}(q(b)).$
 $\text{addcl}(p(a) \mid p(b)).$
 Answer: $\text{addcl}(p(a) \mid p(b)).$
 $\text{call}(p(X)) \& q(X).$
 Answer: $\text{call}(p(a)) \& q(a) \mid \text{call}(p(b)) \& q(b).$

Note that the argument of *call* has an indefinite answer; it is $p(a) \mid p(b)$.

• The PENGUIN system

The PENGUIN system extends STORK to support clausal defaults. The refutation procedure for PENGUIN is based on the method discussed in Section 3.

Syntax of the basic PENGUIN language

The basic PENGUIN alphabet consists of the basic STORK alphabet augmented with the symbol $\underline{I}$. The special symbol $:$ also has a specific function.

The definitions of a PENGUIN *atom*, *variable*, *term*, *atomic formula*, and *clauses* are taken directly from STORK. The concept of a PENGUIN *constant* is extended to incorporate the atoms $\underline{In}$, where n is a number. These atoms represent the typical constants and, as shown in Section 3, cannot be used in programs or queries, being introduced exclusively by the PENGUIN machine.

A PENGUIN *default* is an expression of the form $A:B$, where A is a conjunction of PENGUIN atomic formulas, negated or not, and B is a PENGUIN clause. A PENGUIN *program* is a finite sequence of PENGUIN defaults and PENGUIN clauses, each one ending with a period. A PENGUIN *query* is defined exactly as in STORK. The priority among connectives and the notation for comments are the same as in STORK.

For example, consider the facts "Anne is a student," "John is retired," "If X is retired, then X does not work." Consider the default "Generally, a student does not work." The corresponding PENGUIN program is shown in Box 9.

```

/* Program WORK */
d1: student(Z):¬work(Z).
p1: student(anne).
p2: retired(john).
p3: ¬retired(X)|¬work(X).

(9)

```

```

1. (student(anne),∅,∅)      . originating from p1
2. (retired(john),∅,∅)     . originating from p2
3. (¬retired(x)|¬work(x),∅,∅) . originating from p3
4. (¬work(z),∅,{d1(z)})   . originating from d1
5. (¬work(y),{r1(y)},∅)   . originating from the
                           query

(10)

```

Operational semantics of the basic PENGUIN language

In many aspects, the operation of the PENGUIN machine is very similar to the operation of the STORK machine, described earlier. The strategy adopted by the STORK machine to construct the refutation trees in pre-order is also adopted by the PENGUIN machine, including the order of application of the inference rules and the selection function. A more detailed discussion of PENGUIN can be found in Silva [19].

During the construction of a refutation tree, when the PENGUIN machine finds a failure branch, it immediately tries to fire the defaults. This mechanism does not ensure that answers computed without defaults will be preferentially returned.

The sequence of actions of the PENGUIN machine is briefly described below:

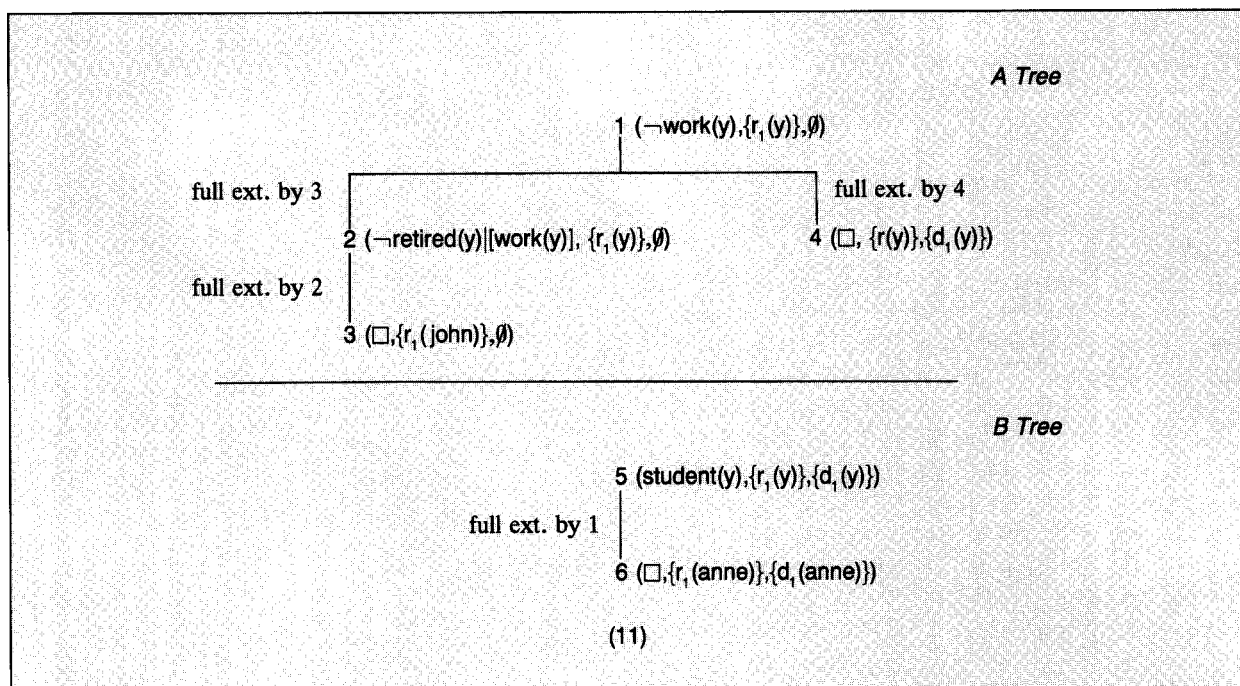
1. Initially, given a PENGUIN program $\Delta = (D, P)$ and a PENGUIN query Q , the machine maps the PENGUIN program clauses directly into a list of indexed and activated elementary chains followed by the set of indexed and activated chains generated from $CL(\neg\exists Q)$ and the list of indexed and activated chains originated from the consequents of the PENGUIN defaults. Thus, the list of chains is composed of chains from the program clauses, chains from the query, and chains from the consequents of the defaults, in this order.
2. The machine constructs a refutation tree whose root is a chain generated from the query. When it finds a successful node, the machine returns the set of defaults used in this refutation. If this set is empty, the machine

stops and the corresponding answer is shown. If the set is not empty, the machine proceeds as follows.

3. The list of chains is now composed only of the chains generated from the PENGUIN clauses followed by the consequents of the PENGUIN defaults. From this list, the machine constructs a refutation tree whose root and initial support is the chain representing the negation of the conjunction of prerequisites of all defaults fired in the previous refutation. Note that this chain in the root is not reused as an auxiliary chain. When it finds a successful branch, the machine returns the set of defaults fired in this refutation, calling itself recursively. If this chain is empty, it is necessary to know whether the refutation sequence is valid. Then, a consistency test is performed, as discussed below. If the test succeeds, a correct answer was computed and it is returned.
4. If there is no successful branch or the consistency test fails, the machine returns **Fail**, causing a backtracking to step 3. After all branches of all possible refutation trees have been tried and no answer is computed, a new chain generated from the query is selected and the process is restarted at step 1.

Observe that the theory corresponding to the program clauses and the consequents of the defaults employed in the refutation must be satisfiable.

The consistency test mentioned above is executed by constructing a set of refutation trees. The roots of the trees are the indexed and activated chains originated from instances of the consequents, i.e., instances obtained by substituting typical constants for variables. The test fails if



the machine finds some success branch and succeeds if all branches of all trees are finite failure branches.

The machine can also produce no answer if it traverses an infinite branch.

For example, let Q be the query $\neg \text{work}(X)$ over the program **WORK**. The indexing and activation of the program and the query results in the sequence shown in Box 10.

The refutation trees are shown in Box 11. The query has two correct answers. The machine will return the first answer detected, i.e., $\neg \text{work}(\text{john})$, which does not use defaults. If required, the machine will return the other correct answer, $\neg \text{work}(\text{anne})$. To compute this last answer, the machine performs the consistency test trying to refute $\neg \text{work}(\text{anne})$ from chains 1, 2, and 3. Since no such refutation is possible, the consistency test succeeds.

PENGUIN extralogical facilities

Defaults bring up new programming aspects, which demand the special extralogical predicates incorporated in PENGUIN. For brevity, we present here only the *def*, *caldef*, and *defanswer* predicates.

The *def(OP)* predicate The extralogical predicate *def(OP)* operates as a switch which allows or prevents the use of defaults in computing an answer to the query. If the argument is *on*, the defaults may be used; if it is *off*, the defaults are disregarded. If the argument is a variable, it returns the current option, *on* or *off*. When a PENGUIN session starts, the option is set to *on*. Note that different

executions of *caldef* generate independent queries and, thus, independent *def(OP)* settings.

The *caldef(C)* predicate The *caldef(C)* predicate has a semantics similar to that of the *call(C)* predicate. The argument C is a conjunction of predicates and works as a subquery. However, unlike the *call(C)* predicate, this subquery may be proved, possibly by considering a different extension of the theory, as a function of the particular defaults corresponding to the program. Thus, the *caldef(C)* isolates the defaults used.

For example, consider the program

true: $a(X)$.
true: $\neg a(X)$.

We can easily see that $a(1)$ and $\neg a(1)$ are correct answers for this program, but they belong to different extensions. Then,

$a(1) \& \text{call}(\neg a(1))$.
 Fail
 $a(1) \& \text{caldef}(\neg a(1))$.
 Answer: $a(1) \& \text{caldef}(\neg a(1))$

The *defanswer(OP)* predicate The extralogical predicate *defanswer(OP)* authorizes or prevents the printing of the defaults used for computing an answer, depending whether the argument *OP* is *on* or *off*. If the argument is a variable, it will be instantiated with the current option of the *defanswer* state. When a PENGUIN session starts, this option is set to *on*.

For example, consider again the same program

b(X).

b(X):a(X).

Then

a(X).

Answer: a(T 0)

defanswer(on)&a(X).

Answer: defanswer(on) & a(T 1)

Defaults: <b(T 1) : a(T 1)>

5. Conclusions

This paper has described the foundations of two logic programming systems, STORK and PENGUIN, with the expressive power of full first-order logic and with a nonmonotonic component, which provide a direct generalization of pure Prolog and can be implemented using the same technology as Prolog processors.

The inference engine of STORK is based on a version of the weak-model elimination method. This refutation method offers an interesting alternative for the construction of logic programming systems because it accepts generic clauses, is input-linear, and does not use factoring, but in spite of these characteristics, maintains completeness. The search space can also be reduced by filters that restrict the application of the inference rules. The refutation procedure for PENGUIN is an extension of weak-model elimination along the lines of Reiter's default logic. It implements the consistency test required by the use of defaults through a strategy quite similar to that adopted to implement negation by finite failure.

STORK and PENGUIN provide more expressive power than Prolog. In particular, they enable the programmer to use classical as well as finite failure negation, choosing the one that best suits his application. Moreover, PENGUIN also allows the representation of default information; that is, it permits carrying on nonmonotonic reasoning using defaults and, possibly, finite failure negation. STORK and PENGUIN are both coded in Prolog and are completely operational.

References

1. D. W. Loveland, "A Simplified Format for the Model Elimination Theorem-Proving Procedure," *J. ACM* **16**, 349-363 (1969).
2. J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, New York, 1986.
3. R. Reiter, "A Logic for Default Reasoning," *Artif. Intell.* **13**, 81-132 (1980).
4. K. L. Clark, "Negation as Failure," *Logic and Databases*, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp. 293-322.
5. M. A. Casanova, R. A. de T. Guerreiro, and A. Silva, "Foundations of Logic Programming based on Model Elimination," *Technical Report CCR073*, Rio Scientific Center, IBM Brazil, 1989.
6. R. A. de T. Guerreiro, A. Silva, and M. A. Casanova, "Computing Answers in Default Logic," *Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence*, Fairfax, VA, 1989, pp. 583-587.
7. A. Silva, R. A. de T. Guerreiro, and M. A. Casanova, "ME-D: A General Clause Logic Programming System with Defaults," *Technical Report CCR093*, Rio Scientific Center, IBM Brazil, 1989.
8. S. Fleising, D. Loveland, A. K. Smiley III, and D. L. Yarmush, "An Implementation of the Model Elimination Proof Procedure," *J. ACM* **21**, 124-139 (1974).
9. M. E. Stickel, "A PROLOG Technology Theorem Prover," *New Generation Computing* **2**, 371-383 (1984).
10. M. A. Casanova and M. E. M. T. Walter, "A Refutation Procedure Based on Model Elimination," *Technical Report CCR043*, Rio Scientific Center, IBM Brazil, 1986.
11. J. W. Lloyd and R. W. Topor, "Making Prolog More Expressive," *J. Logic Program.* **2**, 93-109 (1985).
12. H. Aida, H. Tanaka, and T. Moto-Oka, "A Prolog Extension for Handling Negative Knowledge," *New Generation Computing* **1**, 87-91 (1983).
13. D. L. Poole and R. Goebel, "Gracefully Adding Negation and Disjunction to Prolog," *Proceedings of the Third International Conference on Logic Programming*, London, July 1986, pp. 635-641.
14. K. A. Bowen, "Programming with Full First-Order Logic," *Machine Intell.* **10**, 421-440 (1982).
15. R. Reiter, "On Closed World Databases," *Logic and Databases*, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp. 55-76.
16. C. Green, "Applications of Theorem Proving to Problem Solving," *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, DC, 1969, pp. 219-239.
17. M. A. Casanova, R. A. de T. Guerreiro, and A. Silva, "Logic Programming with General Clauses and Defaults Based on Model Elimination," *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989.
18. R. A. de T. Guerreiro, A. Silva, and M. A. Casanova, "Foundations of Logic Programming with Defaults," (technical report in preparation).
19. A. Silva, "Fundamentos de Programação em Cláusulas Genéricas e Defaults por Eliminação de Modelos," Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil, November 1988.

Received September 28, 1990; accepted for publication November 18, 1991

Ramiro Affonso de Tadeu Guerreiro *IBM Brasil, Industria, Maquinas e Services Ltd., Av. Pasteur 138-146, URCA, Rio de Janeiro, RJ, Brasil (RAMIROG at RIOSYS1, ramirog@riosys1.vnet.ibm.com).* Dr. Guerreiro received a B.Sc. in metallurgical engineering from the Military Institute of Engineering, Brazil, in 1981, an M.Sc. in mathematics from the Pure and Applied Mathematics Institute, IMPA, in 1982, and a D.Sc. in computer science from the Pontifical Catholic University of Rio de Janeiro in 1990. In August 1983 he joined IBM Brazil, where he is now a Marketing Specialist for Customer Information Systems—Latin America. From 1987 to 1991, Dr. Guerreiro was a Researcher at the IBM Rio Scientific Center. His academic interests include logic, artificial intelligence, and database theory; he has published several technical papers in international scientific journals.

Andrea Silva Hemerly *IBM Brasil, Rio Scientific Center, Av. Pres. Vargas, 824/22 andar, 20.071-001, Rio de Janeiro, RJ, Brasil (ANDREASI at RIOVMSC, andrea@vnet.ibm.com).* Ms. Hemerly received a B.Sc. in civil engineering from the University of Espirito Santo, Brazil, and an M.Sc. in computer science from the Pontifical Catholic University of Rio de Janeiro; she is currently completing requirements for a D.Sc. at the Catholic University. In December 1988, she joined the IBM Brazil Rio Scientific Center, where she is now a Researcher. Her academic interests include database, logic programming, and artificial intelligence; she has published several technical papers in international scientific journals.

Marco Antonio Casanova *IBM Brasil, Rio Scientific Center, Av. Pres. Vargas, 824/22 andar, 20.071-001, Rio de Janeiro, RJ, Brasil (CASANOVA at RIOVMSC, casanova@vnet.ibm.com).* Dr. Casanova received a B.Sc. in electronic engineering from the Military Institute of Engineering of Brazil in 1974, an M.Sc. in computer science from the Pontifical Catholic University of Rio de Janeiro in 1976, and a Ph.D. in applied mathematics from Harvard University in 1979. In November 1982, he joined the IBM Brazil Scientific Center, where he is now a Research Manager. From 1980 to 1982, he was Assistant Professor at the Department of Informatics of the Catholic University in Rio, where he also acted as Graduate Program Coordinator during the year 1981–1982. Dr. Casanova is author of the book *The Concurrency Control Problem for Database Systems*, published by Springer-Verlag; he is a co-author of the books (in Portuguese) *Principles of Distributed Database Systems*, *Logic Programming and Prolog*, and *Fundamentals of Multimedia Systems*. He has also published several technical papers in international scientific journals. His academic interests include database theory, database management systems, and logic programming.